main.py

```python
from utils import *

from composer2d import Axises, Cage, Net, Square
from composer3d import Cube, Tetrahedron, Octahedron, Icosahedron, Dodecahedron

from app import Vertex, Object, App


if __name__ == '__main__':
    app = App()

    app.add_debug_object(Object(_position=Vertex(0, 0, 0), _shape=Cage()))
    # app.add_debug_object(Object(_position=Vertex(0, 0, 0), _shape=Net()))

    app.add_object(
        Object(
            _position=Vertex(1, 1, 1),
            _face=Vertex(0, 1, 0),
            _shape=Cube(Vertex(0, 0, 0), 1, COLORS['red']),
            _edge=1
            )
        )
    app.add_object(
        Object(
            _position=Vertex(-1, 0, 0),
            _face=Vertex(0, 1, 0),
            _shape=Tetrahedron(Vertex(0, 0, 0), 1.2, COLORS['green']),
            _edge=1.2
            )
        )
    app.add_object(
        Object(
            _position=Vertex(-2, 0, 0),
            _face=Vertex(0, 1, 0),
            _shape=Octahedron(Vertex(0, 0, 0), .9, COLORS['cyan']),
            _edge=.9
            )
        )
    app.add_object(
        Object(
            _position=Vertex(1, 0, 0),
            _face=Vertex(0, 1, 0),
            _shape=Icosahedron(Vertex(0, 0, 0), .75, COLORS['yellow_c']),
            _edge=.75
            )
        )
    app.add_object(
        Object(
            _position=Vertex(2, 0, 0),
            _face=Vertex(0, 1, 0),
            _shape=Dodecahedron(Vertex(0, 0, 0), .65, COLORS['magenta']),
            _edge=.65
            )
    )

    for shape in app.SHAPES:
```

```
57          shape.push()
58          shape.spin()
59
60      app.run(_debug=True)
```

app.py

```python
from utils import *

from object import Vertex, Shape2D, Shape3D, Object

class App:
    """
    Main class of the project. Initilising window, lights, manages objects and draw them.
    """
    def __init__(self, _resolution : tuple[int, int] = DEFAULT_RESOLUTION) -> None:
        pygame.init()
        pygame.display.set_caption('Plato Solids')
        try:
            pygame.display.set_icon(pygame.image.load('./rsc/icosahedron.png'))
        except:
            try:
                pygame.display.set_icon(pygame.image.load('./icosahedron.png'))
            except:
                pass
        self.display = _resolution
        self.window = pygame.display.set_mode(self.display, DOUBLEBUF | OPENGL)
        gluPerspective(45, (self.display[0] / self.display[1]), 0.1, 50.0)
        glTranslatef(*CAM_POSITION)

        glLight(GL_LIGHT0, GL_POSITION,  (4, 4, 8, 1)) # point light from the left, top, front
        glLightfv(GL_LIGHT0, GL_AMBIENT, (0, 0, 0, 1))
        glLightfv(GL_LIGHT0, GL_DIFFUSE, (1, 1, 1, 1))

        glEnable(GL_DEPTH_TEST)
        # glDepthFunc(GL_LESS)

        self.SHAPES_DEBUG : list[Object] = list()
        self.SHAPES : list[Object] = list()

    def add_debug_object(self, _object : Object) -> None:
        """
        Adds new objects, that are used as debug .
        """
        self.SHAPES_DEBUG.append(_object)

    def add_object(self, _object : Object) -> None:
        """
        Adds new objects, that will be drawn.
        """
        self.SHAPES.append(_object)

    def draw_objects(self, _debug):
        """
        It draws all the objects you added .
        """

        if _debug:
            for shape in self.SHAPES_DEBUG:
                shape.draw(True, False)

        for shape in self.SHAPES:
```

```
56
57              shape.draw(True, True)
58              shape.translate()
59              shape.rotate()
60              shape.collide()
61
62              for _shape in self.SHAPES:
63                  if shape is not _shape:
64                      # print()
65                      def lst_mul(lst, x):
66                          return [a * x for a in lst]
67                      if shape.position.distance(_shape.position) <= (shape.edge +
_shape.edge) * .7:
68                          shape.translation, _shape.translation = lst_mul(shape.translation,
-1), lst_mul(_shape.translation, -1)
69
70              # print(shape)
71
72      def run(self, _debug : bool = False):
73          """
74          Runs project. You can rotate cam Up/Down and Left/Right.
75          """
76          clock = pygame.time.Clock()
77          is_over = False
78
79          global FPS_COUNT
80
81          while not is_over:
82              dX = 0
83              dY = 0
84              dZ = 0
85              angle = 0
86              for event in pygame.event.get():
87                  if event.type == pygame.QUIT or event.type == pygame.K_ESCAPE:
88                      is_over = True
89                      pygame.quit()
90                      quit()
91                  if event.type == pygame.KEYDOWN:
92                      if event.key == pygame.K_UP:
93                          dX += 1
94                          angle = 15
95                      if event.key == pygame.K_DOWN:
96                          dX -= 1
97                          angle = 15
98
99                      if event.key == pygame.K_LEFT:
100                         dY += 1
101                         angle = 15
102                     if event.key == pygame.K_RIGHT:
103                         dY -= 1
104                         angle = 15
105
106                     if event.key == pygame.K_KP7:
107                         dZ += 1
108                         angle = 15
109                     if event.key == pygame.K_KP9:
110                         dZ -= 1
111                         angle = 15
112
113             glRotatef(angle, dX, dY, dZ)
```

```
            # glRotatef(.5, 1, 1, 1)

            FPS_COUNT += 1

            glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)

            glEnable(GL_LIGHT0)
            glEnable(GL_LIGHTING)
            glEnable(GL_COLOR_MATERIAL)

            self.draw_objects(_debug)

            glDisable(GL_LIGHT0)
            glDisable(GL_LIGHTING)
            glDisable(GL_COLOR_MATERIAL)

            clock.tick(FPS_CAP)
            # pygame.display.set_caption(f'{int(clock.get_fps())} [{FPS_COUNT}]')
            pygame.display.flip()

            # pygame.time.wait(30)
            # sleep(2)

if __name__ == '__main__':
    pass
```

object.py

```python
from utils import *
from shape3d import Vertex, Shape2D, Shape3D

class Object:
    def __init__(self,
            _position : Vertex = Vertex(0, 0, 0),
            _face : Vertex = Vertex(0, 0, 1),
            _shape : Shape2D | Shape3D = None,
            _edge : float = 1
            ) -> None:
        self.position : Vertex = _position
        self.face : Vertex = _face
        self.translation : list[float] = None
        self.rotation : tuple[float, str] = None
        self.shape : Shape2D | Shape3D = _shape
        self.edge : float = _edge

    def __repr__(self) -> str:
        return f"POS:  {self.position}\npos:  {self.shape.position}\nFACE: {self.face}\nT: {as_colored(self.translation, fg=FOREGROUND_COLORS_STRONG['yellow'])}\nROT: {as_colored(self.rotation, fg=FOREGROUND_COLORS_STRONG['red'])}\n{self.shape.position - self.shape.vertices[0]}"

    def push(self) -> None:
        """
        Give a random uniform vector of traslation
        """
        self.translation = [
            uniform(*UNIFORM_DIRECTION_AREA),
            uniform(*UNIFORM_DIRECTION_AREA),
            uniform(*UNIFORM_DIRECTION_AREA)
        ]

    def add_translation(self, _vector : list[float]) -> None:
        self.translation = _vector

    def add_rotation(self, _angle : float, _axis : str) -> None:
        self.rotation = (_angle, _axis)

    def draw_vector(self, _vector):
        glLineWidth(2)
        glBegin(GL_LINES)
        glColor3f(*COLORS['cyan'])
        glVertex3f(self.position.x, self.position.y, self.position.z)
        glVertex3f(self.position.x + _vector.x, self.position.y + _vector.y, self.position.z + _vector.z)
        glEnd()
        glLineWidth(1)

    def draw_y_axis(self):
        glLineWidth(2)
        glBegin(GL_LINES)
        glColor3f(*COLORS['red'])
        glVertex3f(self.position.x, self.position.y+1, self.position.z)
        glVertex3f(self.position.x, self.position.y-1, self.position.z)
        glEnd()
        glLineWidth(1)
```

```
54
55      def draw_center(self):
56          glPointSize(8)
57          glBegin(GL_POINTS)
58          glColor3f(*COLORS['white'])
59          glVertex3f(self.position.x, self.position.y, self.position.z)
60          glEnd()
61          glPointSize(1)
62
63          glPointSize(8)
64          glBegin(GL_POINTS)
65          glColor3f(*COLORS['cyan'])
66          glVertex3f(self.position.x + self.shape.position.x, self.position.y +
    self.shape.position.y, self.position.z + self.shape.position.z)
67          glEnd()
68          glPointSize(1)
69
70      def draw(self, _draw_edges : bool = False, _draw_vertices : bool = False) -> None:
71
72          self.shape.draw(self.position, _draw_edges, _draw_vertices)
73
74      def translate(self) -> None:
75          if self.translation:
76              self.position.translate(self.translation)
77
78
79      def rotate(self) -> None:
80          """
81          Rotation in relation to Object Origin (Rotating only the object position and face
    Vector).
82          """
83          if self.rotation:
84              if len(self.rotation) == 2:
85                  self.face.rotate(Vertex(0, 0, 0), *self.rotation)
86                  # self.position.rotate(self.position, *self.rotation)
87                  self.shape.rotate(Vertex(0, 0, 0), *self.rotation)
88              elif len(self.rotation) == 3:
89                  self.face.rotate_any(Vertex(0, 0, 0), *self.rotation)
90                  # self.position.rotate_any(self.position, *self.rotation)
91                  self.shape.rotate_any(Vertex(0, 0, 0), *self.rotation)
92
93
94      def spin(self):
95          self.rotation = [
96              uniform(*UNIFORM_DIRECTION_AREA),
97              uniform(*UNIFORM_DIRECTION_AREA),
98              uniform(*UNIFORM_DIRECTION_AREA)
99          ]
100
101     def collide(self):
102         if self.translation:
103             collision = [1, 1, 1]
104             if self.position.x > BORDER_COLLITION_CAP or self.position.x < -
    BORDER_COLLITION_CAP:
105                 collision[0] = -1
106             if self.position.y > BORDER_COLLITION_CAP or self.position.y < -
    BORDER_COLLITION_CAP:
107                 collision[1] = -1
108             if self.position.z > BORDER_COLLITION_CAP or self.position.z < -
    BORDER_COLLITION_CAP:
109                 collision[2] = -1
```

```
110
111             self.translation = [
112                 self.translation[0] * collision[0],
113                 self.translation[1] * collision[1],
114                 self.translation[2] * collision[2]]
115
116
117 if __name__ == '__main__':
118     pass
119
```

shape3d.py

```python
from utils import *
from shape2d import Vertex, Shape2D

class Shape3D:
    def __init__(self, _position : Vertex = Vertex(0, 0, 0), _vertices : list[Vertex] = list(), _surfaces : list[Shape2D] = list(), _color : tuple[float] = None, _direction : list[float] = None) -> None:
        self.position : Vertex = _position
        self.vertices : list[Vertex] = _vertices
        self.surfaces : list[Shape2D] = _surfaces
        self.color: tuple[float] = _color
        self.direction : list[int] = [uniform(*UNIFORM_DIRECTION_AREA), uniform(*UNIFORM_DIRECTION_AREA), uniform(*UNIFORM_DIRECTION_AREA)] if not _direction else _direction

    def darken_color(self, _color, _delta = 0.2):
        return (_color[0] - _delta, _color[1] - _delta, _color[2] - _delta)

    def draw(self, _object_position : Vertex, _draw_edges : bool = False, _draw_vertices : bool = False) -> None:
        for surface in self.surfaces:
            if _draw_edges:
                for i in range(len(surface.vertices) - 1):
                    glLineWidth(2)
                    glBegin(GL_LINES)
                    glColor3f(*self.darken_color(surface.color))
                    glVertex3f(
                        _object_position.x + surface.vertices[i].x,
                        _object_position.y + surface.vertices[i].y,
                        _object_position.z + surface.vertices[i].z
                    )
                    glVertex3f(
                        _object_position.x + surface.vertices[i + 1].x,
                        _object_position.y + surface.vertices[i + 1].y,
                        _object_position.z + surface.vertices[i + 1].z
                    )
                    glEnd()
                    glLineWidth(1)
            surface.draw(_object_position, _draw_vertices)

    def translate(self, _vector : list[float]) -> None:
        for vertex in self.vertices:
            vertex.translate(_vector)
        self.position.translate(_vector)

    def rotate(self, _object_position : Vertex, _angle : float, _axis : str) -> None:
        for vertex in self.vertices:
            vertex.rotate(_object_position, _angle, _axis)
        self.position.rotate(_object_position, _angle, _axis)


    def rotate_any(self, _object_position : Vertex, _yaw : float, _pitch : float, _roll : float) -> None:
        for vertex in self.vertices:
            vertex.rotate_any(_object_position, _yaw, _pitch, _roll)
        self.position.rotate_any(_object_position, _yaw, _pitch, _roll)

```

composer3d.py

```python
from utils import *
from shape3d import Vertex, Shape2D, Shape3D

def Cube(_position : Vertex = Vertex(0, 0, 0), _edge : float = 1, _color : tuple[float] = None):
    """
    Initialize 3D Cube object .
    """
    half_edge = _edge / 2
    '''
            3           2
        0           1

            7           6
        4           5
    '''
    VERTICES = [
        Vertex(
            _position.x - half_edge,
            _position.y - half_edge,
            _position.z - half_edge,
        ),
        Vertex(
            _position.x - half_edge,
            _position.y + half_edge,
            _position.z - half_edge,
        ),
        Vertex(
            _position.x + half_edge,
            _position.y + half_edge,
            _position.z - half_edge,
        ),
        Vertex(
            _position.x + half_edge,
            _position.y - half_edge,
            _position.z - half_edge,
        ),

        Vertex(
            _position.x - half_edge,
            _position.y - half_edge,
            _position.z + half_edge,
        ),
        Vertex(
            _position.x - half_edge,
            _position.y + half_edge,
            _position.z + half_edge,
        ),
        Vertex(
            _position.x + half_edge,
            _position.y + half_edge,
            _position.z + half_edge,
        ),
        Vertex(
            _position.x + half_edge,
            _position.y - half_edge,
```

```
            _position.z + half_edge,
        ),

    ]
    return Shape3D(_position, VERTICES, [
        Shape2D(_position, [
            VERTICES[0],
            VERTICES[1],
            VERTICES[2],
            VERTICES[3],
        ], _color),
        Shape2D(_position, [
            VERTICES[4],
            VERTICES[5],
            VERTICES[6],
            VERTICES[7],
        ], _color),

        Shape2D(_position, [
            VERTICES[0],
            VERTICES[1],
            VERTICES[5],
            VERTICES[4],
        ], _color),
        Shape2D(_position, [
            VERTICES[2],
            VERTICES[3],
            VERTICES[7],
            VERTICES[6],
        ], _color),

        Shape2D(_position, [
            VERTICES[0],
            VERTICES[3],
            VERTICES[7],
            VERTICES[4],
        ], _color),
        Shape2D(_position, [
            VERTICES[1],
            VERTICES[2],
            VERTICES[6],
            VERTICES[5],
        ], _color),

    ])


def Tetrahedron(_position : Vertex = Vertex(0, 0, 0), _edge : float = 1, _color :
tuple[float] = None):
    """
    Initialize 3D Tetrahedron object .
    """
    half_edge = _edge / 2
    y_base = _edge / 4 * ((2 / 3) ** (1 / 2))
    z_diff = _edge / (2 * (3 ** (1 / 2)))
    '''
                3

                2
        0           1
```

```
    '''
    VERTICES = [
        Vertex(
            _position.x - half_edge,
            _position.y - y_base,
            _position.z - z_diff,
        ),
        Vertex(
            _position.x + half_edge,
            _position.y - y_base,
            _position.z - z_diff,
        ),
        Vertex(
            _position.x,
            _position.y - y_base,
            _position.z + 2 * z_diff,
        ),
        Vertex(
            _position.x,
            _position.y + 3 * y_base,
            _position.z,
        ),

    ]
    return Shape3D(_position, VERTICES, [
        Shape2D(_position, [
            VERTICES[0],
            VERTICES[1],
            VERTICES[2],
        ], _color),
        Shape2D(_position, [
            VERTICES[0],
            VERTICES[1],
            VERTICES[3],
        ], _color),
        Shape2D(_position, [
            VERTICES[1],
            VERTICES[2],
            VERTICES[3],
        ], _color),
        Shape2D(_position, [
            VERTICES[2],
            VERTICES[0],
            VERTICES[3],
        ], _color),

    ])


def Octahedron(_position : Vertex = Vertex(0, 0, 0), _edge : float = 1, _color :
tuple[float] = None):
    """
    Initialize 3D Octahedron object .
    """
    half_edge = _edge / 2
    height = _edge * ((1 / 2) ** (1 / 2))
    '''
                4

            3           2
```

```
        0           1

                5
    '''
    VERTICES = [
        Vertex(
            _position.x - half_edge,
            _position.y,
            _position.z - half_edge,
        ),
        Vertex(
            _position.x + half_edge,
            _position.y,
            _position.z - half_edge,
        ),
        Vertex(
            _position.x + half_edge,
            _position.y,
            _position.z + half_edge,
        ),
        Vertex(
            _position.x - half_edge,
            _position.y,
            _position.z + half_edge,
        ),
        Vertex(
            _position.x,
            _position.y + height,
            _position.z,
        ),
        Vertex(
            _position.x,
            _position.y - height,
            _position.z,
        ),

    ]
    return Shape3D(_position, VERTICES, [
        Shape2D(_position, [
            VERTICES[0],
            VERTICES[1],
            VERTICES[4],
        ], _color),
        Shape2D(_position, [
            VERTICES[0],
            VERTICES[1],
            VERTICES[5],
        ], _color),

        Shape2D(_position, [
            VERTICES[1],
            VERTICES[2],
            VERTICES[4],
        ], _color),
        Shape2D(_position, [
            VERTICES[1],
            VERTICES[2],
            VERTICES[5],
        ], _color),

```

```
        Shape2D(_position, [
            VERTICES[2],
            VERTICES[3],
            VERTICES[4],
        ], _color),
        Shape2D(_position, [
            VERTICES[2],
            VERTICES[3],
            VERTICES[5],
        ], _color),

        Shape2D(_position, [
            VERTICES[3],
            VERTICES[0],
            VERTICES[4],
        ], _color),
        Shape2D(_position, [
            VERTICES[3],
            VERTICES[0],
            VERTICES[5],
        ], _color),

    ])


def Icosahedron(_position : Vertex = Vertex(0, 0, 0), _edge : float = 1, _color :
tuple[float] = None):
    """
    Initialize 3D Icosahedron object .
    """
    fi = pi * 2 / 5
    r = _edge / (2 * sin(fi / 2))
    y_top = _edge / 2 * (3 - 1 / sin(fi / 2)) ** (1 / 2)
    circumradius = ((10 + (2 * 5 ** (1 / 2))) ** (1 / 2)) * (_edge / 4)
    print(circumradius, y_top)
    '''
                5
                3
        4               2
            0       1

                6
        7               10
            8       9
                11

    '''
    VERTICES = list()
    for i in range(5):
        vertex = Vertex(
            _position.x,
            _position.y + circumradius - y_top,
            _position.z - r,
        )
        vertex.rotate(_position, fi * i, 'Ro_y')
        VERTICES.append(vertex)
    VERTICES.append(
        Vertex(
            _position.x,
            _position.y + circumradius,
```

```
            _position.z,
        ))


    for i in range(5):
        vertex = Vertex(
            _position.x,
            _position.y - circumradius + y_top,
            _position.z + r,
        )
        vertex.rotate(_position, fi * i, 'Ro_y')
        VERTICES.append(vertex)
    VERTICES.append(
        Vertex(
            _position.x,
            _position.y - circumradius,
            _position.z,
        ))

    return Shape3D(_position, VERTICES, [
        Shape2D(_position, [
            VERTICES[0],
            VERTICES[1],
            VERTICES[5],
        ], _color),
        Shape2D(_position, [
            VERTICES[1],
            VERTICES[2],
            VERTICES[5],
        ], _color),
        Shape2D(_position, [
            VERTICES[2],
            VERTICES[3],
            VERTICES[5],
        ], _color),
        Shape2D(_position, [
            VERTICES[3],
            VERTICES[4],
            VERTICES[5],
        ], _color),
        Shape2D(_position, [
            VERTICES[4],
            VERTICES[0],
            VERTICES[5],
        ], _color),

        Shape2D(_position, [
            VERTICES[0],
            VERTICES[1],
            VERTICES[9],
        ], _color),
        Shape2D(_position, [
            VERTICES[1],
            VERTICES[2],
            VERTICES[10],
        ], _color),
        Shape2D(_position, [
            VERTICES[2],
            VERTICES[3],
            VERTICES[6],
```

```
        ], _color),
        Shape2D(_position, [
            VERTICES[3],
            VERTICES[4],
            VERTICES[7],
        ], _color),
        Shape2D(_position, [
            VERTICES[4],
            VERTICES[0],
            VERTICES[8],
        ], _color),

        Shape2D(_position, [
            VERTICES[6],
            VERTICES[7],
            VERTICES[3],
        ], _color),
        Shape2D(_position, [
            VERTICES[7],
            VERTICES[8],
            VERTICES[4],
        ], _color),
        Shape2D(_position, [
            VERTICES[8],
            VERTICES[9],
            VERTICES[0],
        ], _color),
        Shape2D(_position, [
            VERTICES[9],
            VERTICES[10],
            VERTICES[1],
        ], _color),
        Shape2D(_position, [
            VERTICES[10],
            VERTICES[6],
            VERTICES[2],
        ], _color),

        Shape2D(_position, [
            VERTICES[6],
            VERTICES[7],
            VERTICES[11],
        ], _color),
        Shape2D(_position, [
            VERTICES[7],
            VERTICES[8],
            VERTICES[11],
        ], _color),
        Shape2D(_position, [
            VERTICES[8],
            VERTICES[9],
            VERTICES[11],
        ], _color),
        Shape2D(_position, [
            VERTICES[9],
            VERTICES[10],
            VERTICES[11],
        ], _color),
        Shape2D(_position, [
            VERTICES[10],
```

```
            VERTICES[6],
            VERTICES[11],
        ], _color),
    ])


def Dodecahedron(_position : Vertex = Vertex(0, 0, 0), _edge : float = 1, _color :
tuple[float] = None):
    """
    Initialize 3D Dodecahedron object .
    """
    fi = pi * 2 / 5
    r = _edge / (2 * sin(fi / 2))
    circumradius = (3 ** (1 / 2)) * (1 + (5 ** (1 / 2))) * _edge / 4
    _y_top = _edge / 2 * (3 * (3 + 2 * (5 ** (1 / 2))) / 2 - 1 / sin(fi / 2)) ** (1 / 2)

    y_top = (circumradius ** 2 - r ** 2) ** (1 / 2)
    d = _edge / 2 * (1 + (5 ** (1 / 2)))
    r_ = d / (2 * sin(fi / 2))
    y_mid = (circumradius ** 2 - r_ ** 2) ** (1 / 2)


    # print(circumradius, y_top)
    '''

                3
        4               2
            0       1

            8       7
        9               6
                5


    '''
    VERTICES = list()

    # top
    for i in range(5):
        vertex = Vertex(
            _position.x,
            _position.y + y_top,
            _position.z - r,
        )
        vertex.rotate(_position, fi * i, 'Ro_y')
        VERTICES.append(vertex)

    # top-mid
    for i in range(5):
        vertex = Vertex(
            _position.x,
            _position.y + y_mid,
            _position.z - r_,
        )
        vertex.rotate(_position, fi * i, 'Ro_y')
        VERTICES.append(vertex)

    # bottom-mid
    for i in range(5):
        vertex = Vertex(
```

```
            _position.x,
            _position.y - y_mid,
            _position.z + r_,
        )
        vertex.rotate(_position, fi * i, 'Ro_y')
        VERTICES.append(vertex)

    # bottom
    for i in range(5):
        vertex = Vertex(
            _position.x,
            _position.y - y_top,
            _position.z + r,
        )
        vertex.rotate(_position, fi * i, 'Ro_y')
        VERTICES.append(vertex)

    return Shape3D(_position, VERTICES, [
        Shape2D(_position, [
            VERTICES[0],
            VERTICES[1],
            VERTICES[2],
            VERTICES[3],
            VERTICES[4],
        ], _color),

        Shape2D(_position, [
            VERTICES[0],
            VERTICES[1],
            VERTICES[6],
            VERTICES[-7],
            VERTICES[5],
        ], _color),
        Shape2D(_position, [
            VERTICES[1],
            VERTICES[2],
            VERTICES[7],
            VERTICES[-6],
            VERTICES[6],
        ], _color),
        Shape2D(_position, [
            VERTICES[2],
            VERTICES[3],
            VERTICES[8],
            VERTICES[-10],
            VERTICES[7],
        ], _color),
        Shape2D(_position, [
            VERTICES[3],
            VERTICES[4],
            VERTICES[9],
            VERTICES[-9],
            VERTICES[8],
        ], _color),
        Shape2D(_position, [
            VERTICES[4],
            VERTICES[0],
            VERTICES[5],
            VERTICES[-8],
            VERTICES[9],
```

```
        ], _color),

        Shape2D(_position, [
            VERTICES[-1],
            VERTICES[-2],
            VERTICES[-7],
            VERTICES[6],
            VERTICES[-6],
        ], _color),
        Shape2D(_position, [
            VERTICES[-2],
            VERTICES[-3],
            VERTICES[-8],
            VERTICES[5],
            VERTICES[-7],
        ], _color),
        Shape2D(_position, [
            VERTICES[-3],
            VERTICES[-4],
            VERTICES[-9],
            VERTICES[9],
            VERTICES[-8],
        ], _color),
        Shape2D(_position, [
            VERTICES[-4],
            VERTICES[-5],
            VERTICES[-10],
            VERTICES[8],
            VERTICES[-9],
        ], _color),
        Shape2D(_position, [
            VERTICES[-5],
            VERTICES[-1],
            VERTICES[-6],
            VERTICES[7],
            VERTICES[-10],
        ], _color),

        Shape2D(_position, [
            VERTICES[-1],
            VERTICES[-2],
            VERTICES[-3],
            VERTICES[-4],
            VERTICES[-5],
        ], _color),
    ])

```

shape2d.py

```python
from vertex import Vertex
from utils import *

class Shape2D:
    def __init__(self, _position : Vertex = Vertex(0, 0, 0), _vertices : list[Vertex] = list(), _color : tuple[float] = None, _render_mode = GL_TRIANGLE_FAN) -> None:
        self.position : Vertex = _position
        self.vertices : list[Vertex] = _vertices
        self.color = _color
        self.render_mode = _render_mode

    def draw(self, _object_position : Vertex, _true, _draw_vertices : bool = False) -> None:
        glBegin(self.render_mode)
        glColor3f(*self.color)
        for vertex in self.vertices:
            glVertex3f(
                _object_position.x + vertex.x,
                _object_position.y + vertex.y,
                _object_position.z + vertex.z)
            if _draw_vertices:
                vertex.draw()
        glEnd()

    def translate(self, _vector : list[float]) -> None:
        for vertex in self.vertices:
            vertex.translate(_vector)

    def rotate(self, _object_position : Vertex, _angle : float, _axis : str) -> None:
        for vertex in self.vertices:
            vertex.rotate(_object_position, _angle, _axis)

    def rotate_any(self, _object_position : Vertex, _yaw : float, _pitch : float, _roll : float) -> None:
        for vertex in self.vertices:
            vertex.rotate_any(_object_position, _yaw, _pitch, _roll)

```

composer2d.py

```python
from utils import *
from shape2d import Vertex, Shape2D

def Line(_position : Vertex = Vertex(0, 0, 0), _edge : float = 1, _color : tuple[float] = None, _rotation : tuple[float, str] = None):
    """
    Initialize 2D Line object .
    """
    half_edge = _edge / 2
    VERTICES = [
        Vertex(
            _position.x,
            _position.y,
            _position.z - half_edge,
        ),
        Vertex(
            _position.x,
            _position.y,
            _position.z + half_edge,
        ),

    ]
    return Shape2D(_position, VERTICES, _color, GL_LINES)


def Axises():
    """
    Initialize 2D Axises object .
    """
    half_edge = 4
    VERTICES = [
        Vertex(
            0,
            0,
            0,
        ),
        Vertex(
            0,
            0,
            -half_edge,
        ),

        Vertex(
            0,
            0,
            0,
        ),
        Vertex(
            0,
            0,
            half_edge,
        ),

        Vertex(
            0,
            0,
```

```
            0,
        ),
        Vertex(
            0,
            -half_edge,
            0,
        ),

        Vertex(
            0,
            0,
            0,
        ),
        Vertex(
            0,
            half_edge,
            0,
        ),

        Vertex(
            0,
            0,
            0,
        ),
        Vertex(
            -half_edge,
            0,
            0,
        ),

        Vertex(
            0,
            0,
            0,
        ),
        Vertex(
            half_edge,
            0,
            0,
        ),

    ]
    return Shape2D(Vertex(0, 0, 0), VERTICES, COLORS['green'], GL_LINES)


def Net(_size : int = 4):
    """
    Initialize 2D Net (every 1 unit = line) object .
    """
    len = _size * 2
    VERTICES = []
    for z in range(-_size, _size+1):
        for a in range(-_size, _size+1):
            VERTICES.append(Vertex(a, -_size, z))
            VERTICES.append(Vertex(a, +_size, z))

            VERTICES.append(Vertex(-_size, a, z))
            VERTICES.append(Vertex(+_size, a, z))

    for y in range(-_size, _size+1):
```

```
        for x in range(-_size, _size+1):
            VERTICES.append(Vertex(x, y, -_size))
            VERTICES.append(Vertex(x, y, +_size))

    return Shape2D(Vertex(0, 0, 0), VERTICES, COLORS['g_cage'], GL_LINES)


def Cage(_size : int = 4):
    """
    Initialize 2D Cage (Borders) object .
    """
    VERTICES = [
        Vertex(
            -_size,
            -_size,
            -_size,
        ),
        Vertex(
            +_size,
            -_size,
            -_size,
        ),
        Vertex(
            -_size,
            -_size,
            -_size,
        ),
        Vertex(
            -_size,
            +_size,
            -_size,
        ),
        Vertex(
            -_size,
            -_size,
            -_size,
        ),
        Vertex(
            -_size,
            -_size,
            +_size,
        ),

        Vertex(
            -_size,
            +_size,
            -_size,
        ),
        Vertex(
            -_size,
            +_size,
            +_size,
        ),
        Vertex(
            -_size,
            +_size,
            -_size,
        ),
        Vertex(
            +_size,
```

```
            +_size,
            -_size,
        ),

        Vertex(
            +_size,
            +_size,
            +_size,
        ),
        Vertex(
            -_size,
            +_size,
            +_size,
        ),
        Vertex(
            +_size,
            +_size,
            +_size,
        ),
        Vertex(
            +_size,
            -_size,
            +_size,
        ),
        Vertex(
            +_size,
            +_size,
            +_size,
        ),
        Vertex(
            +_size,
            +_size,
            -_size,
        ),

        Vertex(
            +_size,
            -_size,
            +_size,
        ),
        Vertex(
            -_size,
            -_size,
            +_size,
        ),
        Vertex(
            +_size,
            -_size,
            +_size,
        ),
        Vertex(
            +_size,
            -_size,
            -_size,
        ),

        Vertex(
            -_size,
            -_size,
            +_size,
```

```
        ),
        Vertex(
            -_size,
            +_size,
            +_size,
        ),
        Vertex(
            +_size,
            -_size,
            -_size,
        ),
        Vertex(
            +_size,
            +_size,
            -_size,
        ),

    ]

    return Shape2D(Vertex(0, 0, 0), VERTICES, COLORS['g_cage'], GL_LINES)


def Square(_position : Vertex = Vertex(0, 0, 0), _edge : float = 1, _color : tuple[float] = None, _rotation : tuple[float, str] = None):
    """
    Initialize 2D Square object .
    """
    half_edge = _edge / 2
    VERTICES = [
        Vertex(
            _position.x - half_edge,
            _position.y - half_edge,
            _position.z,
        ),
        Vertex(
            _position.x + half_edge,
            _position.y - half_edge,
            _position.z,
        ),
        Vertex(
            _position.x + half_edge,
            _position.y + half_edge,
            _position.z,
        ),
        Vertex(
            _position.x - half_edge,
            _position.y + half_edge,
            _position.z,
        )
    ]
    square = Shape2D(_position, VERTICES, _color)
    if _rotation is not None:
        square.rotate(*_rotation)
    return square


def Cirlce(_position : Vertex = Vertex(0, 0, 0), _radius : float = 1, _n_sides : int = 100, _color : tuple[float] = None):
    """
    Initialize 2D Cirlce object .
```

```
    """
    angle = 2 * pi / _n_sides
    VERTICES = []
    for i in range(_n_sides):
        vertex = Vertex(
            _position.x,
            _position.y + _radius,
            _position.z,
        )
        vertex.rotate(_position, i * angle, 'Ro_z')
        VERTICES.append(vertex)
    return Shape2D(_position, VERTICES, _color)
```

matrix.py

```python
class Matrix:
    def __init__(self, _content : list[list[float]] | tuple[float], _is_vector : bool = False) -> None:
        if _is_vector:
            self.V = _content
            self.m = 1
            self.n = len(self.V)
        else:
            self.M = _content
            self.m = len(self.M)
            self.n = len(self.M[0])
        self.is_vector = _is_vector

    def to_vertex(self) -> tuple[float, float, float]:
        if self.is_vector:
            return self.V

    def mul(self, v1, v2):
        return sum([v1[i] * v2[i] for i in range(len(v1))])

    def T(self):
        content = [[0 for _ in range(self.m)] for _ in range(self.n)]
        for m in range(self.m):
            for n in range(self.n):
                content[n][m] = self.M[m][n]
        return Matrix(content)



    def __mul__(self, _mul):
        if self.is_vector and _mul.is_vector and self.n == _mul.n:
            return self.mul(self.V, _mul.V)
        elif not self.is_vector and _mul.is_vector:
            return Matrix(tuple([self.mul(row, _mul.V) for row in self.M]), True)
        elif not self.is_vector and not _mul.is_vector and self.n == _mul.m:
            content = []
            _mul_t = _mul.T()
            for line_a in self.M:
                row = []
                for line_b in _mul_t.M:
                    row.append(self.mul(line_a, line_b))
                content.append(row)
            return Matrix(content)


    def __repr__(self) -> str:
        if self.is_vector:
            return self.V.__repr__()
        else:
            ret = '[\n'
            for row in self.M:
                ret += f'    {row}\n'
            ret += ']'
            return ret


```

```python
Identity3_Matrix = Matrix(
    [
        [1, 0, 0],
        [0, 1, 0],
        [0, 0, 1]
    ]
)

Zero3_Matrix = Matrix(
    [
        [0, 0, 0],
        [0, 0, 0],
        [0, 0, 0]
    ]
)

Zero3_Vector = Matrix((0, 0, 0), True)

from math import cos, sin

M = {
    'T'     : lambda _vector :  Matrix([
                                    [1, 0, 0, _vector[0]],
                                    [0, 1, 0, _vector[1]],
                                    [0, 0, 1, _vector[2]],
                                    [0, 0, 0, 1],
                                ]),
    'Ro_x'  : lambda _angle :   Matrix([
                                    [1, 0, 0],
                                    [0, cos(_angle), -sin(_angle)],
                                    [0, sin(_angle), cos(_angle)],
                                ]),
    'Ro_y'  : lambda _angle :   Matrix([

                                    [cos(_angle), 0, sin(_angle)],
                                    [0, 1, 0],
                                    [-sin(_angle), 0, cos(_angle)],
                                ]),
    'Ro_z'  : lambda _angle :   Matrix([
                                    [cos(_angle), -sin(_angle), 0],
                                    [sin(_angle), cos(_angle), 0],
                                    [0, 0, 1],
                                ]),
    'Sc'    : Zero3_Matrix,
    'Re_x'  : Zero3_Matrix,
    'Re_y'  : Zero3_Matrix,
    'Re_z'  : Zero3_Matrix,
}

def Ro_Any_Matrix(_yaw : float, _pitch : float, _roll : float) -> Matrix:
    return M['Ro_z'](_yaw) * M['Ro_y'](_pitch) * M['Ro_x'](_roll)

if __name__ == '__main__':
    print(Ro_Any_Matrix(1, 1, 1))


```

vertex.py

```python
from utils import *
from matrix import Matrix, M, Ro_Any_Matrix

class Vertex:
    def __init__(self, _x : float, _y : float, _z : float) -> None:
        self.x : float = _x
        self.y : float = _y
        self.z : float = _z
        self.V : tuple[float, float, float] = self.set_tuple()

    def set_tuple(self) -> tuple[float, float, float]:
        return (self.x, self.y, self.z)

    def length(self) -> float:
        return (self.x ** 2 + self.y ** 2 + self.z ** 2) ** (1 / 2)

    def distance(self, _v) -> float:
        return ((self.x - _v.x) ** 2 + (self.y - _v.y) ** 2 + (self.z - _v.z) ** 2) ** (1 / 2)

    def draw(self) -> None:
        glBegin(GL_POINTS)
        glVertex3f(self.x, self.y, self.z)
        glEnd()

    def translate(self, _vector : list[float]) -> list[float]:
        new_vertex : Matrix = M['T'](_vector) * Matrix([*self.V, 1], _is_vector=True)
        self.x = new_vertex.V[0]
        self.y = new_vertex.V[1]
        self.z = new_vertex.V[2]
        self.V = new_vertex.V

    def rotate(self, _position, _angle : float, _axis : str) -> None:
        new_vertex : Matrix = M[_axis](_angle) * Matrix((self - _position).V, _is_vector=True)
        self.x = new_vertex.V[0] + _position.x
        self.y = new_vertex.V[1] + _position.y
        self.z = new_vertex.V[2] + _position.z
        self.V = new_vertex.V

    def rotate_any(self, _position, _yaw : float, _pitch : float, _roll : float) -> None:
        new_vertex : Matrix = Ro_Any_Matrix(_yaw, _pitch, _roll) * Matrix((self - _position).V, _is_vector=True)
        self.x = new_vertex.V[0] + _position.x
        self.y = new_vertex.V[1] + _position.y
        self.z = new_vertex.V[2] + _position.z
        self.V = new_vertex.V

    def __sub__(self, _v):
        return Vertex(self.x - _v.x, self.y - _v.y, self.z - _v.z)

    def __mul__(self, _mul):
        if isinstance(_mul, (int, float)):
            return Vertex(self.x * _mul, self.y * _mul, self.z * _mul)

    def __repr__(self) -> str:
        return f'({self.x}, {self.y}, {self.z})'
```

console.py

```python
def clear_console() -> None :
    from os import system, name
    if name == 'nt':
        _ = system('cls')
    else:
        _ = system('clear')

def clear_screen() -> None :
    print('\x1b[H\x1b[3J', end='')
    print('\n'*64)
    print('\x1b[H\x1b[3J', end='')


RESET_COLOR = '0'

FOREGROUND_COLORS = {
    'black'     : '30',
    'red'       : '31',
    'green'     : '32',
    'yellow'    : '33',
    'blue'      : '34',
    'purple'    : '35',
    'cian'      : '36',
    'white'     : '37'
}
FOREGROUND_COLORS_STRONG = {
    'black'     : '90',
    'red'       : '91',
    'green'     : '92',
    'yellow'    : '93',
    'blue'      : '94',
    'purple'    : '95',
    'cian'      : '96',
    'white'     : '97'
}

BACKGROUND_COLORS = {
    'on_black'  : '40',
    'on_red'    : '41',
    'on_green'  : '42',
    'on_yellow' : '43',
    'on_blue'   : '44',
    'on_purple' : '45',
    'on_cian'   : '46',
    'on_white'  : '47'
}
BACKGROUND_COLORS_STRONG = {
    'on_black'  : '100',
    'on_red'    : '101',
    'on_green'  : '102',
    'on_yellow' : '103',
    'on_blue'   : '104',
    'on_purple' : '105',
    'on_cian'   : '106',
    'on_white'  : '107'
}
```

```python

def set_color(fg : str = None, bg : str = None) -> None:
    if fg:
        print(f"\x1b[{fg}m", end='')
    if bg:
        print(f"\x1b[{bg}m", end='')

def reset_color() -> None:
    print(f"\x1b[{RESET_COLOR}m", end='')

def colored(*msgs : tuple[str], fg : str = None, bg : str = None) -> None:
    set_color(fg, bg)
    print(*msgs)
    reset_color()

def as_colored(text : str, fg : str = RESET_COLOR, bg : str = RESET_COLOR) -> str :
    return f"\x1b[{fg};{bg}m{text}\x1b[0m"


# def error(msg : str) -> None :
#   print(f"\x1b[{FOREGROUND_COLORS['red']}m{msg}\x1b[0m")

def error(*msgs : tuple[str]) -> None:
    colored(*msgs, fg=FOREGROUND_COLORS_STRONG['red'])

def as_error(text : str) -> str :
    return as_colored(text, fg=FOREGROUND_COLORS_STRONG['red'])

# def alert(msg : str) -> None :
#   print(f"\x1b[{FOREGROUND_COLORS['yellow']}m{msg}\x1b[0m")

def alert(*msgs : tuple[str]) -> None:
    colored(*msgs, fg=FOREGROUND_COLORS_STRONG['yellow'])

def as_alert(text : str) -> str :
    return as_colored(text, fg=FOREGROUND_COLORS_STRONG['yellow'])


if __name__ == '__main__':
    abc = 123
    print(abc)
    print(*[1, 2, 3, 4, 5])

    alert(abc)
    alert(abc, 'abc')

    error(abc)
    error(abc, 'abc')

```

utils.py

```python
from math import pi, sin, cos, radians
from random import randint, random, uniform
from time import sleep

import pygame
from pygame.locals import *
from OpenGL.GL import *
from OpenGL.GLU import *

from console import *

DEFAULT_RESOLUTION = (800, 800)
FPS_CAP = 30
FPS_COUNT = 0
CAM_POSITION = (0, 0, -14)
# LIGHT_POSITION = -CAM_POSITION[2]
LIGHT_CAP = 4.0
LIGHT_POSITION = (0, 0, LIGHT_CAP, 1)
LIGHT_DIFFUSION = (LIGHT_CAP, LIGHT_CAP, LIGHT_CAP, 1)

UNIFORM_DIRECTION_CAP = 0.1
UNIFORM_DIRECTION_AREA = (-UNIFORM_DIRECTION_CAP, UNIFORM_DIRECTION_CAP)
BORDER_COLLITION_CAP = 4.0

COLORS = {
    'white'     :   (1, 1, 1),
    'gray'      :   (.75, .75, .75),
    'dark_gray' :   (.5, .5, .5),
    'black'     :   (0, 0, 0),

    'red'       :   (1, 0, 0),
    'green'     :   (0, 1, 0),
    'g_cage'    :   (0, 1, 0),
    'blue'      :   (0, 0, 1),

    'magenta'   :   (1, 0, 1),
    'yellow'    :   (1, 1, 0),
    'yellow_c'  :   (1, 0.75, 0),
    'cyan'      :   (0, 1, 1),
}
```